How to Correctly Deal With Pseudorandom Numbers in Manycore Environments - Application to GPU programming with Shoverand

Jonathan Passerat-Palmbach, David R.C. Hill

UMR 6158 CNRS

How to Correctly Deal With Pseudorandom Numbers in Manycore Environments - Application to GPU programming with Shoverand

Jonathan Passerat-Palmbach* † ‡ § ¶ ,
David R.C. Hill † ‡ § ¶

Originally published in: Proceedings of the IEEE High Performance Computing and Simulation conference 2012 — July 2012 — pp 25-31
http://dx.doi.org/10.1109/HPCSim.2012.6266887
(tutorial paper)

Abstract: Stochastic simulations are often sensitive to the source of randomness that characterizes the statistical quality of their results. Consequently, we need highly reliable Random Number Generators (RNGs) to feed such applications. Recent developments try to shrink the computation time by relying more and more General Purpose Graphics Processing Units (GP-GPUs) to speed-up stochastic simulations. Such devices bring new parallelization possibilities, but they also introduce new programming difficulties. Since RNGs are at the base of any stochastic simulation, they also need to be ported to GP-GPU. There is still a lack of well-designed implementations of quality-proven RNGs on GP-GPU platforms. In this paper, we introduce ShoveRand, a framework defining common rules to generate random numbers uniformly on GP-GPU. Our framework is designed to cope with any GPU-enabled development platform and to expose a straightforward interface to users. We also provide an existing RNG implementation with this framework to demonstrate its efficiency in both development and ease of use.

Keywords: pseudorandom numbers, manycore, GP-GPU, High Performance Computing, Shoverand, stochastic simulation

* This author's PhD is partially funded by the Auvergne Regional Council and the FEDER
† ISIMA, Institut Supérieur d'Informatique, de Modélisation et de ses Appplications, BP 10125, F-63173 AUBIERE
‡ Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
§ Clermont Université, Université Blaise Pascal, BP 10448, F-63000 CLERMONT-FERRAND
¶ CNRS, UMR 6158, LIMOS, F-63173 AUBIERE

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

1 Introduction

More than 50% of High Performance Computing (HPC) applications now require the use of Random Number Generators (RNGs). We have had RNGs adapted to intensive parallel computing at our disposal for more than a decade . But even though the subject is well mastered by specialists like Pierre L'Ecuyer or Makoto Matsumoto, this is not the case for many colleagues who are acknowledged as specialists in other domains but are not aware of the recent parallelization techniques than can be used with modern generators. Even if we now also have access to true random numbers with quantum devices, they are still clumsy to produce in parallel with satisfying rates and costs. In addition, such devices are still subject to partial breakdowns and even the latest ones do have some very little biases that have to be corrected (such processing is sometimes included inside the device). In addition, stochastic experiments have to be reproducible, not only for debugging purposes but also for variance reduction techniques, for sensitivity analysis and for many other statistical techniques [12]. This implies storing the sequences (in order to be able to reproduce exactly the same sequence several times). This last point leads to the fact that we cannot provide enough "true" numbers for many High Performance Computing applications such as parallel Monte Carlo for nuclear medicine [13, 6]. Consequently, software random number generation remains the prevailing method for HPC, and specialists have been warning us for years to be particularly careful when dealing with parallel stochastic simulations [5, 8, 26].

Recent developments try to shrink computation time by relying more and more on General Purpose Graphics Processing Units (GP-GPUs) to speed-up stochastic simulations. Such devices bring new parallelization possibilities, but they also introduce new programming difficulties. Since the introduction of Tesla boards, Nvidia, ATI and other manufacturers of GP-GPUs have changed the way we use our high computing performance resources. Since 2010, we have seen that the top supercomputers are now often hybrid. Since RNGs are at the base of any stochastic simulation, they also need to be ported to GP-GPU. There is still a lack of well-designed implementations of quality-proven RNGs on GP-GPU platforms. Consequently, we also need a survey of the current Pseudo Random Numbers Generators (PRNG) available on GPU. We will discuss the recent Mersenne Twister for Graphics Processors (MTGP) that was initially released in 2010, but also more recent generators with cryptographic inspiration that have been presented lately at the annual supercomputing conference.

In this paper we intend to present the good practices when dealing with pseudorandom streams in parallel. After explaining the kind of problems we may encounter, the state of the art in terms of RNGs, testing suite and parallelization techniques is presented. Then, GPU considerations and hybrid computing constraints are exposed. Next, the Shoverand framework that facilitates a safe usage of pseudorandom streams for modern GPU hardware is presented, and we also show how to integrate your preferred PRNG in Shoverand if your RNG of choice is not already included in the framework.

2 Using Parallel Random Streams

2.1 Parallel stochastic simulation

Parallel and Distributed Simulation (PDS) is an extensive research domain where effective solutions have been developed. Deterministic communications protocols for synchronous simulation and asynchronous simulation have been studied to avoid deadlocks and to preserve the causality and determinism principles. When dealing with stochastic simulations, random numbers should be generated in parallel in order for each Processing Element (PE) to be able to autonomously get its own independent random number stream. If such autonomy is not guaranteed, the parallelism

is affected [2].

2.2 State of the art in terms of sequential generators

The most famous sequential generator currently available is Mersenne Twister (MT hereafter) [19]. Since the initial proposition at the end of the nineties, a whole family of MT generators has been proposed. SFMT, a member of this family, is an SIMD-oriented version of the original MT generator [32]. SFMT proposed the following improvements: speed (twice as fast as MT), a better equidistribution, a quicker recovery from bad initialization (zero-excess in the initial state) and even an increased period length (ranging from $2^{607}-1$ to $2^{216091}-1$). A GP-GPU version of MT (named MTGP) was also proposed by Saito and Matsumoto; it comes with companion software for parallelization (MTGPDC) [33]. Based on similar principles (Generalized Feedback Shift Register), Panneton, in collaboration with L'Ecuyer and Matsumoto, proposed the WELL generators with even better statistical properties [22].

Pierre L'Ecuyer suggests that multiple recurrence generators (MRGs) with much smaller periods (above 2^{100} but under 2^{200}) like MRG32k3a [16] can suffice for modern applications. This generator comes with very interesting statistical properties and is easy to parallelize. Smaller periods also authorize fast mathematical parallelization techniques. The MT family now proposes TinyMT which can benefit of this point. In 2011, Salmon et al recently introduced statistically sound counter based pseudorandom generators with relatively small periods [34]. These proposals led to very interesting implementations on GP-GPUs.

With this large set of 'good' sequential RNGs, the main question is: how can we make a safe RNG repartition in order to keep, on the one hand, efficiency, and on the other hand a sound statistical quality of the simulation in order to obtain reliable results.

2.3 Partitionning techniques

Assigning random sequences to parallel processing elements (PEs) can be done with one of the two following approaches. The first one proposes to partition a unique random stream, whereas the second approach deals with multiple independent streams. In this case, independent streams are obtained by parameterizing a family of generators.

When dealing with the partitioning of a unique random source we find the following variants in literature. The Central Server approach runs a single PRNG and provides on demand pseudorandom numbers to different PEs. We can also cite Boolean cellular automata since they have been considered to generate parallel pseudorandom numbers, but we are not aware of any recent high performance stochastic simulations using this technique. Another technique is known as the Sequence Splitting method, sometimes named Blocking or Regular Spacing. The principle behind those names is simple: the main sequence is split into 'N' non-overlapping contiguous blocks. Until recently, the computing of a jump in the sequence was unavailable for modern generators with linear recurrences modulo 2 and huge periods (such as the families of MT & WELL generators). This problem was solved in [7], but despite its relative efficiency, it is still considered as slow by specialists (compared to what can be obtained for MRG32k3a with a smaller period). The Indexed Sequence, or Random Spacing Technique, is another technique that consists in initializing the same generator with different random statuses. The last known technique to partition a unique stream is known as the Leap Frog Technique. With this method, random numbers are distributed to processing elements like a deck of cards dealt to card players.

When a single stream is partitioned into many different substreams, we talk about Parameterization. This technique issues many instances from the same family of generators. Independent generators, with different parameters, are used for each processing element (PE). For the

Mersenne Twister family this technique is named Dynamic Creator (DC) and generates by parameterization highly independent Mersenne Twister generators. Our best mathematicians state that PNRGs based on linear recurrences defined by matrices with characteristic polynomials relatively prime to each other are supposed to be highly independent. In addition to the original MT generator, TinyMT and MTGP for GP-GPUs benefit from the same approach. Even though it is considered safe according to the scientific community, we recommend testing the resulting generators. In case of failure we can give feedback to authors and this was recently done for MTGP [25].

The interested reader can find the details of the different approaches in the following surveys [9, 10], the latter being the most complete reference (a full journal paper with the latest updates), with discussions around advantages and drawbacks of each technique.

2.4 Testing sequential and paralell RNGs

The previous 'fine' generators were found reliable thanks to thorough statistical and empirical testing. Testing techniques were initially proposed by Knuth and the first testing software that became famous in the nineties was the DieHard testing suite proposed by Marsaglia. Another famous Statistical Testing Suite is proposed by the National Institute for Standards and Technology (NIST STS) and is particularly appreciated when cryptographic qualities are required. [1] developed the DieHarder suite as an update of Marsaglia's work. Rütti and Troyer also presented a testing suite with Petersen [30]. But currently, the most complete collection of utilities for the empirical and statistical testing of uniform random number generators is in our opinion TestU01. This software library proposed by [17] is now the reference for most scientists of the domain.

For parallel testing, Srinivasan and Mascagni gave interesting advice [36] based on the tests proposed by Mascagni in the Scalable library for Pseudorandom Number Generation (SPRNG) in 1997. Approximately at the same time, [2] proposed a set of techniques for empirical testing of parallel random number generators. We have to be aware that we do not have at our disposal mathematical theorems or techniques to explicitly give a proof of independence between generators or between generated parallel random streams. To avoid long-range correlations we can have a look at interesting propositions made by [4, 5].

2.5 Software aid for the distribution of random streams

The reference software library dealing with parallelization of pseudorandom numbers is SPRNG (Scalable library for Pseudorandom Number Generation) [18]. As stated before, SPRNG also proposes a small test suite. The Dynamic Creator discussed above and proposed initially by Matsumoto and Nishimura is a software aid to provide mutually independent Mersenne twister generators for parallel computing, which was published two years later [20]. L'Ecuyer and his team proposed a package able to produce many long streams and substreams in C, C++, Java, and FORTRAN [16] and also in R in a later version. Coddington and Newell proposed the JaPara library (a Java Parallel Random Number Library) for High-Performance Computing [3]. If we consider Java, our advice would be to use the SSJ package, still from L'Ecuyer's team. It provides the basic structures for handling multiple streams, with efficient methods to move streams around [14].

We also proposed higher level software tools to help in the distribution of stochastic simulations on local clusters and institutional computing Grids. The DistMe toolkit [28, 29] and the OpenMole software based on declarative task delegation [27] helps removing the burden of random streams distribution.

In the first part of this paper, we have given a shallow survey of distribution techniques,

libraries and tools. The second part focuses on Shoverand: our library proposal integrating the guidelines introduced in the previous lines.

3 Purpose of Shoverand

As we have seen in the previous parts of this article, it is very important to deal carefully with pseudorandom numbers distribution when working in parallel environments such as GPUs. Still we cannot ask any user that wants to leverage GPUs' computational power in his stochastic simulations to be aware of every theoretical consideration that will prevent his simulation results from being biased. Thus, we introduced Shoverand [23], a framework that provides Pseudorandom Number Generation (PRNG) facilities to CUDA-enabled GPU applications.

Shoverand combines several aspects to ease developments of stochastic-enabled applications on GPU. First, its API is quite similar to what can be encountered when using high-level CPU languages like C++ or Java. Second, Shoverand's main goal is to handle the distribution of stochastic streams automatically without any intervention from the user. Finally, our framework also targets PRNG developers: indeed, Shoverand only integrates third-party PRNGs and focuses on unifying their interface. To do so, we integrate compile-time constraints that check whether the algorithm meets our guidelines.

4 PRNGs embedded in Shoverand

At the time of writing, Shoverand embeds several PRNGs. All these algorithms have been selected according to their intrinsic properties. We first consider their statistical properties in a sequential environment, because a PRNG could not cope with the requirements of parallel environments if its sequential version was poor. Consequently, every PRNG wrapped in Shoverand must satisfy the most stringent testing battery currently available, namely BigCrush from TestU01 [17]. PRNGs that pass all those tests are referred to as "Crush-resistant" in [34]. While being Crush-resistant cannot ensure a perfect randomness of the considered pseudorandom stream, it is a satisfying property that few PRNGs can be proud of.

Additionally, the retained algorithms must support a reliable technique to distribute numbers in a parallel environment. We have previously surveyed such techniques in [10], but only some of them can be applied on a GPU platform [24]. The chosen ones are then ideal candidates to be ported to GPU, if not available yet, and moreover to be integrated in Shoverand. We detail hereafter the PRNGs that are currently, or will soon be, embedded in Shoverand.

4.1 MRG32k3a

Introduced by Pierre L'Ecuyer in [15], MRG32k3a is particularly suited to parallelization among small computational elements such as threads thanks to its intrinsic properties. This PRNG's lightweight data structure only stores 6 integers to handle its state. The algorithm itself is quite short, and relies on simple operations to issue new random numbers. The parameters chosen for MRG32k3a are such that it has a full period of 2^{191} numbers. This period is fairly enough since L'Ecuyer suggests that periods between 2^{100} and 2^{200} are highly sufficient even for large-scale simulations. MRG32k3a has been designed to produce independent streams and sub-streams from its original random sequence thanks to its parameters that enable safe Sequence Splitting [10]. The internal parameters split the initial sequence into 2^{64} adjacent streams of 2^{127} random numbers, themselves divided into sub-streams containing 2^{76} elements.

Now considering the distribution aspect, we can assign a stream or a sub-stream to each computational element according to the Sequence Splitting technique. As long as we are focusing on parallel applications that are CUDA-enabled, we are dealing with fine-grained Single Instruction, Multiple Threads (SIMT) applications. It means that the computational elements are, in our case, the logical threads of a CUDA kernel and the principle of SIMT is to load the device with as much threads as possible. Still, we do not expect having to deal with more than 2^{64} parallel threads, which is the total number of independent streams bearing 2^{127} random numbers each that MRG32k3a can provide.

4.2 TinyMT

TinyMT is the latest offspring from the Mersenne Twister family. TinyMT is not described in any scientific article yet, but information about it can be found on its dedicated webpage [31]. This PRNG matches the requirements we have formulated for a PRNG to be integrated into Shoverand: it is described as producing a good quality output, according to TestU01 statistical tests, and displays a long-enough period of 2^{127} numbers. TinyMT leverages parameterization to provide highly independent streams, each stream being represented by a unique parameterized status.

The theoretical aspect of this approach is very satisfying, but TinyMT parameterized statuses need to be precomputed by a piece of software called Dynamic Creator (DC), which is shipped with the PRNG as an open-source binary. The idea here is to initialize each computing elements with a different status, since DC can create over $2^{32} \times 2^{16}$ independent statuses. However, memory footprint considerations forced us to propose a hybrid implementation where the same independent parameterized status is shared among all the threads of a CUDA block. Independence between random sources is achieved by feeding each thread with a sub-stream of the original stream, following the Sequence Splitting technique. To do so, the original stream is sliced in equal chunks whose starting point, the seed status, is indicated to threads depending on their identifier. As a consequence, each thread will always consume the same random sequence in different replications of the same execution, thus ensuring reproducibility of the experience.

4.3 Philox and Threefry

Philox and Threefry are counter-based PRNGs [34] also relying on parameterization to solve random streams partition concerns. Like any other PRNG considered in this study, they are Crush-resistant and display good performance in regards to their low memory footprint and high numbers throughput. They appear to be better suited than TinyMT (or any other member of the Mersenne Twister family) to a straightforward GPU implementation since their parameters are formed by a single key that can be set at runtime according to each thread's unique identifier. Please note that the GPU implementation of these PRNGs is directly provided by their authors. Both CUDA and OpenCL implementations are proposed for Philox and Threefry.

5 Case study: generating pseudorandom numbers in a CUDA kernel with Shoverand

In this section, we describe a major aspect of Shoverand: its user-friendly interface. We will see that on both host and device sides, our API is very expressive while remaining quite concise. Shoverand competes with two major counterparts in the CUDA world, both coming from an NVIDIA initiative, named Thrust [11] and cuRand [21]. These two libraries are also providing

random number generation features but vary from Shoverand on several points that we will compare in this section.

5.1 Host side: Initialization phase

From the end-user's point of view, Shoverand requires an initialization phase in order to allocate its internal data structures on the device and perform some initializations. As a matter of fact, depending on the chosen PRNG, initialization might involve external data to be read from a parameter file, as is the case with TinyMT for instance.

We previously saw that we needed to consider the distribution technique and the PRNG algorithm as a pair. As a consequence, distribution techniques vary from one PRNG to another in Shoverand, but their initialization phase require the same data, which is basically the number of CUDA blocks that the kernel using the PRNG will spawn. This data being stored prior to the kernel call, no superfluous parameter needs to be passed to the kernel. This feature allows users not to have their hands tied when designing their kernels, since Shoverand does not impact kernels' prototypes like other libraries do.

As a result, the host side initialization phase boils down to a single call to a static method named init, which must be provided by every PRNG implementation to satisfy Shoverand's rules.

5.2 Device side: Computation phase

Using the device side of Shoverand is even simpler than the host side. You only have to create an instance of the PRNG you want to use and let its class' constructor do the rest. Device side initializations are performed behind the scenes by the constructor so that users have nothing to do. Then, you can pick random numbers by calling the next() method on the previously created object. If you are used to random number generation facilities offered by high-level languages such as Java, making use of Shoverand in your kernel is really intuitive.

5.3 Comparison with Thrust and cuRand

Thrust and cuRand are two projects developed by NVIDIA fellows. While cuRand is part of the CUDA SDK, Thrust is an external open-source library that can be downloaded from an Internet repository. In the paper originally introducing Shoverand [23], we had surveyed these two libraries and identified their major drawbacks that led us to design Shoverand. The following lines investigate the changes brought by the state-of-the-art versions of Thrust and cuRand.

cuRand is NVIDIA's solution to random number generation on GPU. In [23], we mentioned that cuRand suffered from the poor statistical quality of the PRNGs it embedded. The last version of this library partially solves this problem by following the advice we made in our previous paper. Now, cuRand embeds renowned high-quality PRNGs such as MRG32k3a [15] and MTGP [33], whose pseudorandom streams are stated as "Crush-resistant".

On the other hand, cuRand's API remains poor and forces users to add extra parameters in the prototypes of every kernel taking advantage of the library. The C API is not generic and loses its consistency when non-default options are enabled: for instance, MTGP's initialization step is achieved through a dedicated call in cuRand that is totally irrelevant when used with another PRNG. This approach is not convenient when you want to quickly swap PRNGs to study the impact of various random sources on a given application.

Thrust is an open source GP-GPU-enabled general purpose library developed by NVIDIA fellows. This project aims at providing a GP-GPU-enabled library equivalent to some classic general-purpose C++ libraries, like STL or Boost. Classes are split through several namespaces,

such as Thrust::random. The latter contains all classes and methods related to random numbers generation on GP-GPU. Thrust::random implements three PRNGs, each through a different C++ class template. We find a Linear Congruential Generator (LCG), a Linear Feedback Shift (LFS) and a Subtract With Carry (SWC), which are widely reckoned as not adapted to High Performance Computing.

Still, Thrust offers a nice API that mirrors Boost's. The random namespace provides user-friendly features very close to Shoverand's abilities. For instance, neither explicit initializations nor parameters are required in order to benefit from random number generation facilities in a kernel.

As a conclusion, we have on the one hand cuRand, a library that is improving after having been criticized by the research community for the statistical characteristics of its embedded PRNGs, and that exposes a restrictive API; and on the other hand, we have Thrust and its nice "à la Boost" API, yet powered by poor quality PRNGs. Then, Shoverand is based upon the good achievements from these two libraries: indeed, it exposes a user-friendly API while integrating only Crush-resistant PRNGs.

Listing 1 shows off how to pick up pseudorandom numbers from Shoverand. In this code snippet, we consider both the initialization on the host side and the random number generation on the device side:

Listing 1: Example of use of Shoverand's API

```
using shoverand::RNG;
using shoverand::MRG32k3a;

// kernel using Shoverand
__global__ void fooKernel(float* ddata) {

        RNG< float, MRG32k3a >          rng;

        ddata[blockDim.x * blockIdx.x + threadIdx.x] = rng.next();
}

...

RNG< float, MRG32k3a >::init(block_num);

fooKernel <<< block_num, thread_num >>>(d_data);

RNG< float, MRG32k3a >::release();
```

6 Case study: embedding a new PRNG into Shoverand

Shoverand is not only a library but also a framework that allows PRNG developers to insert their own proposals as long as they follow some rules. In order to help them in their task, Shoverand employs a mechanism called concept checking that lets us express constraints in Shoverand's code that will be checked at compile-time for all the classes that inherit from ours. Concept checking is a generic programming feature available through different implementations with C++. It was introduced by [35] and implemented in the Boost Concept Check Library (BCCL). The application scope of concept checking is quite wide and goes from interface checking (i.e., verifying

the presence of a given method within a class), to assessing the presence of a particular member from a given type in a class.

Such a mechanism forces developers to match our interface without having to introduce costly runtime techniques like polymorphism and consequently virtual methods.

Thanks to the BCCL, we are able to design constraints that will make any compilation attempt fail if they are not met. In Shoverand, we force every PRNG algorithm class to inherit from our base RNG class. Then, each user-defined subclass must define at least 3 methods: init() and release(), which deal with parameters allocation and initialization from the host side, and next(), which picks up the next random number within a kernel, on the device side. In the same way, BCCL also permits us to verify that developers have provided two members to their class: the seed and parameterized statuses. These two members respectively represent the current state of the PRNG and its initial parameters. Fig. 1 sketches a UML class diagram of the expected content of a PRNG class to be embedded in Shoverand:

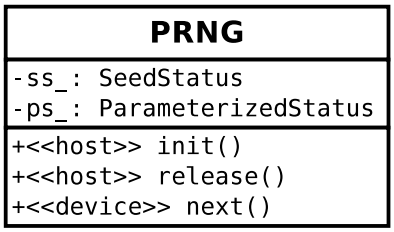

Figure 1: UML class diagram of the expected interface of a PRNG in Shoverand

7 Conclusion

In this paper, we demonstrate how parallel stochastic simulations can be seriously impacted by the quality of the underlying Random Number Generators (RNGs), and the partitioning techniques used to feed such applications [10]. Still, we have at our disposal a set of very reliable sequential generators and a set of distribution techniques adapted to the different kinds of generators.

From a practitioner's point of view, we have described Shoverand: a CUDA library that embeds the PRNGs displaying the best statistical quality. Moreover, its user-friendly interface make it a very serious choice when faced to its counterparts.

Shoverand is freely available for download at the following URL: `http://http://forge.clermont-universite.fr/projects/shoverand`.

Acknowledgements

The authors would like to thank Luc Touraille for his careful reading and useful suggestions on the draft. This work received financial support from the Auvergne Regional Council.

References

[1] Robert G. Brown, Dirk Eddelbuettel, and David Bauer. Dieharder: A random number test suite, 2006.

[2] Paul D. Coddington and Sung-Hoon Ko. Random number generator for parallel computers. In *Proceedings of the 12th international conference on Supercomputing*, pages 282–288. ACM New York, NY, USA, 1998. ISBN: 089791998X.

[3] P.D. Coddington and A.J. Newell. Japara-a java parallel random number generator library for high-performance computing. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004.

[4] A. De Matteis and S. Pagnutti. Parallelization of random number gemerators and long-range correlations. *Numerische Mathematik*, 53(5):595–608, 1988.

[5] A. De Matteis and S. Pagnutti. Controlling correlations in parallel monte carlo. *Parallel Computing*, 21(1):73–84, 1995.

[6] Z. El Bitar, D. Lazaro, C. Coello, V. Breton, D. Hill, and I. Buvat. Fully 3d monte carlo image reconstruction in spect using functionalregions. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators,Spectrometers, Detectors and Associated Equipment*, 569(2):399–403, 2006.

[7] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for f2-linear random number generators. *INFORMS Journal on Computing*, 20(3):385–390, Summer 2008.

[8] Peter Hellekalek. Don't trust parallel monte carlo! In *Proceedings of Parallel and Distributed Simulation PADS98*, pages 82–89. IEEE, 1998.

[9] David R. C. Hill. Practical distribution of random streams for stochastic high performance computing. In *IEEE International Conference on High Performance Computing & Simulation (HPCS 2010)*, pages 1–8, 2010. invited paper.

[10] David R.C. Hill, Claude Mazel, and Jonathan Passerat-Palmbach. Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2012. To be published.

[11] Jared Hoberock and Nathan Bell. Thrust: A parallel template library. `http://www.meganewtons.com/`, 2010. Version 1.3.0.

[12] J.P.C. Kleijnen. *Statistical tools for simulation practitioners*. Marcel Dekker, Inc., 1986.

[13] D. Lazaro, Z.E. Bitar, V. Breton, D. Hill, and I. Buvat. Fully 3d monte carlo reconstruction in spect: a feasibility study. *Physics in Medicine and Biology*, 50:3739, 2005.

[14] P. L'Ecuyer and E. Buist. Simulation in java with ssj. In *Proceedings of the 37th conference on Winter simulation*, pages 611–620. Winter Simulation Conference, 2005.

[15] Pierre L'Ecuyer. Good parameters and implementations for combined multiple recursive generators. *Operations Research*, 47(1):159–164, 1999.

[16] Pierre L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, November-December 2002.

[17] Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22:1–40, August 2007.

[18] M. Mascagni. Some methods of parallel pseudorandom number generation. *IMA Volumes in Mathematics and Its Applications*, 105:277–288, 1999.

[19] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*, 8(1):3–30, January 1998.

[20] Makoto Matsumoto and Takuji Nishimura. Dynamic creation of pseudorandom number generators. In Harald Niederreiter and Jerome Spanier, editors, *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69. Springer, 2000.

[21] NVIDIA. *CUDA CURAND Library*. NVDIA Corporation, February 2012. v1.6.0.

[22] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear reccurences modulo 2. *ACM Transactions on Mathematical Software*, 32:1–16, 2006.

[23] Jonathan Passerat-Palmbach, Claude Mazel, Bruno Bachelet, and David R. C. Hill. Shoverand: a model-driven framework to easily generate random numbers on gp-gpu. In *IEEE International Conference on High Performance Computing & Simulation*, pages 41–48. IEEE, 2011.

[24] Jonathan Passerat-Palmbach, Claude Mazel, and David R.C. Hill. Pseudo-random streams for distributed and parallel stochastic simulations on gp-gpu. *Journal of Simulation*, 2012. to be published.

[25] Jonathan Passerat-Palmbach, Claude Mazel, Antoine Mahul, and David R. C. Hill. Reliable initialization of gpu-enabled parallel stochastic simulations using mersenne twister for graphics processors. In *Europeans Simulation and Modeling Conference 2010*, pages 187–195, October 2010. ISBN: 978-90-77381-57-1.

[26] K. Pawlikowski. Towards credible and fast quantitative stochastic simulation. In *Proceedings of International SCS Conference on Design, Analysis andSimulation of Distributed Systems, DASD*, volume 3, 2003.

[27] R. Reuillon, F. Chuffart, M. Leclaire, T. Faure, N. Dumoulin, and D. Hill. Declarative task delegation in openmole. In Waleed W. Smari, editor, *Proceedings of the 2010 International Conference on High Performance Computing and Simulation*, pages 55–61, 2010.

[28] R. Reuillon, DRC Hill, Z. El Bitar, and V. Breton. Rigorous distribution of stochastic simulations using the distmetoolkit. *Nuclear Science, IEEE Transactions on*, 55(1):595–603, 2008.

[29] Romain Reuillon, Mamadou K. Traore, Jonathan Passerat-Palmbach, and David R.C. Hill. Parallel stochastic simulations with rigorous distribution of pseudo-random numbers with distme: Application to life science simulations. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2011.

[30] M. Rütti, M. Troyer, and W.P. Petersen. A generic random number generator test suite. *Arxiv preprint math/0410385*, 2004.

[31] Mutsuo Saito. Tinymt, 2011.

[32] Mutsuo Saito and Makoto Matsumoto. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In Alexander Keller, Stefan Heinrich, and Harald Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, volume 2, pages 607–622. Springer Berlin Heidelberg, 2008.

[33] Mutsuo Saito and Makoto Matsumoto. Variants of mersenne twister suitable for graphic processors. arXiv:1005.4973v3, 2012. submitted.

[34] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 16:1–16:12, New York, NY, USA, 2011. ACM.

[35] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in c++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, 2000.

[36] A. Srinivasan, M. Mascagni, and D. Ceperley. Testing parallel random number generators. *Parallel Computing*, 29(1):69–94, 2003.

UMR 6158 CNRS

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Publisher
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France
http://limos.isima.fr/